**Laboratorio #3**

**AndresEmilioQuintoVillagran 18288**

Fecha de Entrega: 23 de marzo, 2022.

Descripción: en este laboratorio se empleará *multithreading* por medio de pthreads y OpenMP para desarrollar un verificador de soluciones para *sudokus* de nueve por nueve. Los entregables serán todo el código escrito, así como un documento que responda a las preguntas planteadas al final. Se recomienda auxiliar sus respuestas con *screenshots* de la ejecución de su programa.

Materiales: una máquina virtual con Linux o Windows, pero que tenga GCC, y documentación sobre OpenMP.

Contenido:

OpenMP busca paralelizar (no sólo ejecutar de forma concurrente), por lo que su funcionamiento requiere de más de un procesador. Si usará una máquina virtual, comience por asegurarse de que su máquina cuente con cuatro procesadores (en VirtualBox, ventana *Settings*, menú *System*, *tab Processor*). La cantidad se especifica para uniformizar las respuestas a las preguntas que se plantean al final.

Cree un programa en C llamado SudokuValidator.c. En él escriba tres funciones que se encarguen de revisar que todos los números del uno al nueve estén:

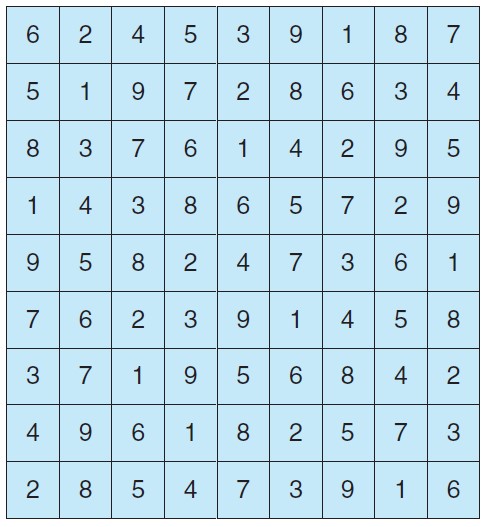
* En cada columna de un arreglo de nueve por nueve.
* En cada fila de un arreglo de nueve por nueve.
* En un subarreglo de tres por tres dentro de un arreglo de nueve por nueve.

Las funciones para verificación de filas y columnas serán iguales exceptuando un intercambio de índices al recorrer el arreglo. La función de revisión de subarreglos debe recibir una fila y una columna para ubicar la esquina superior izquierda de un cuadrado de tres por tres, donde iniciará la revisión dentro del arreglo de nueve por nueve. Todas estas funciones se deben basar en ciclos for obligatoriamente.

Este programa recibirá, en terminal, la ubicación de un archivo (sólo el nombre, si está en el mismo directorio que SudokuValidator.c) que contiene una solución a un *sudoku* de nueve por nueve. El formato de las soluciones debe ser un único *string* de ochenta y un dígitos, en la primera línea, comenzando por la celda superior izquierda del *sudoku* y avanzando de izquierda a derecha, por filas. Para este laboratorio se provee una solución de ejemplo en el archivo “*sudoku”*.

Lo primero que su main() deberá hacer es abrir el archivo usando open() y *mappearlo* a su memoria usando mmap(). Luego debe ejecutar un for en el que se copie cada símbolo del *string* en el archivo de solución a un arreglo bidimensional de nueve por nueve, de modo que le quede una grilla lógica como la que se muestra en la página siguiente.

Se recomienda que su arreglo bidimensional sea global (es decir, que esté declarado fuera del main()) para que sea accesible por varios *threads*. Luego de llenar la grilla, escriba un for que haga la revisión, con su función, de los subarreglos de tres por tres que conforman el arreglo de nueve por nueve (**nota**: revise los subarreglos de tres por tres cuya primera posición (si comenzamos desde 1) sea [𝑖, 𝑖] para 𝑖 ∈ {1,4,7}).



*Grilla lógica ejemplar*

Luego de lo anterior, obtenga el número de proceso (no el de *thread*) y ejecute un fork(). En el proceso hijo convierta el número del proceso padre (no el de *thread*) a texto, y ejecute por medio de execlp() el siguiente comando:

# ps –p <#proc> -lLf

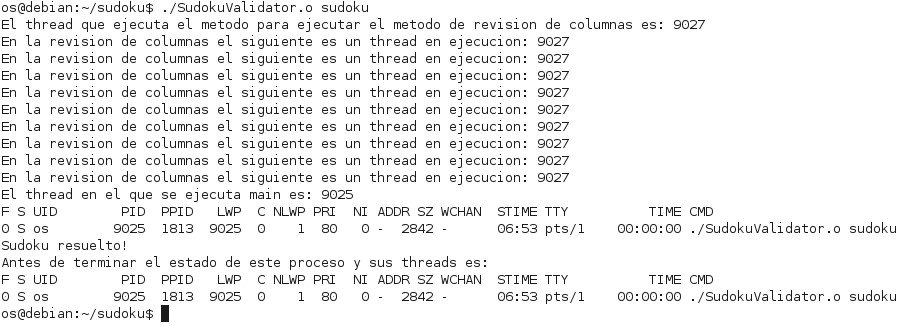
donde <#proc> es el número del proceso padre. Este comando permite ver información relacionada al proceso <#proc> que incluye los *lightweight processes* que tenga asociados.

En el proceso padre:

* Cree un pthread que haga su revisión de columnas.
* Ejecute pthread\_join() y luego despliegue el número de *thread* en ejecución. Para lograrlo debe #incluir <sys/syscall.h> en su programa y ejecutar syscall(SYS\_gettid) (el resultado de esta llamada de sistema es el *id* del *thread*).
* Espere a que concluya el hijo que está ejecutando ps.
* Realice su revisión de filas.
* Despliegue si la solución al *sudoku* es válida o no.
* Ejecute un nuevo fork() y ejecute el comando ps en el proceso hijo, tal como se describe en instrucciones anteriores. Esto servirá para comparar el número de LWP’s asociados al proceso padre cuando se está realizando la revisión de columnas y cuando (el padre) está a punto de terminar.
* Espere al hijo y retorne 0.

Observe que la creación de un *thread* que ejecute la revisión de columnas implica la creación de una función que sea asignable a un *thread* en el cual, a su vez, se ejecute su función de revisión de columnas. Es decir, una función que tenga tipo de retorno void\* y que termine con pthreads\_exit(0). En esa función tipo void\* también despliegue el número de *thread* en ejecución.

El siguiente es un ejemplo de cómo podría verse el *output* de su programa hasta este momento:



R//

A picture containing text

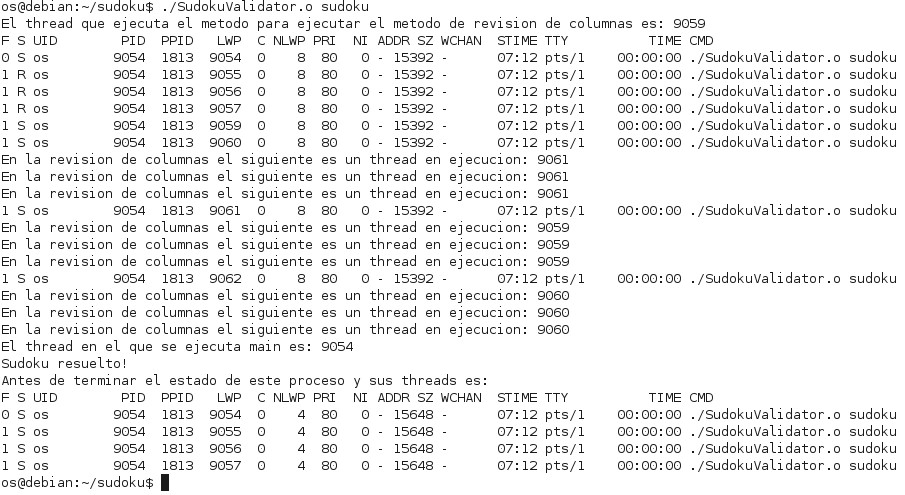
Description automatically generated

Como siguiente paso deberá paralelizar todos los ciclos for que pueda (vea la nota **importante**) usando OpenMP. Para ello simplemente es necesario que la siguiente línea preceda inmediatamente a la del for en cada caso:

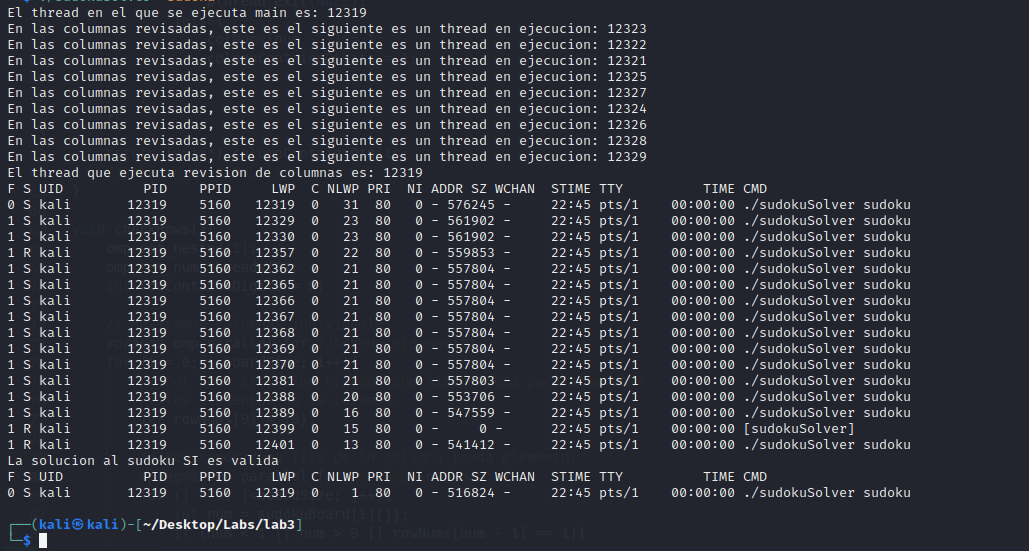
# #pragma omp parallel for

**Importante**: evite las *race conditions*. Investigue el uso de la directiva private de OpenMP para auxiliarse en este aspecto. No todos los ciclos for deberán ser precedidos por la directiva.

Ejecutar su programa ahora deberá resultar en un *output* similar al siguiente:



R//



Anote el número de LWP’s que se tienen durante la revisión de columnas y antes de terminar el programa.

Agregue la siguiente instrucción al principio de main():

# omp\_set\_num\_threads(1);

Ejecute su programa y note el resultado de las ejecuciones de ps. También anote los números de *thread* desplegados durante la revisión de columnas.

Ahora, agregue la siguiente directiva a todas las líneas #pragma que incluyó anteriormente:

# schedule(dynamic)

A picture containing graphical user interface

Description automatically generated

Ejecute su programa varias veces y observe los números de *thread* que se despliegan durante la revisión de columnas. Compárelos con el resultado de ps que se despliega durante la ejecución del pthread y anote sus observaciones.

Como siguiente paso, agregue una llamada a omp\_set\_num\_threads() al inicio de cada función donde se ejecute un for paralelo, determinando el número de *threads* adecuados (*e.g.*, si su función ejecuta un for paralelo de nueve iteraciones, posiblemente el número de *threads* deba ser nueve). Ejecute su programa varias veces y anote los efectos sobre los *threads* en los resultados de ps. Repita el procedimiento comentando la cláusula schedule() en el primer for paralelo de su revisión de columnas. Finalmente agregue la siguiente instrucción al principio de cada función que use OpenMP:

# omp\_set\_nested(true);

Ejecute su programa y anote los efectos sobre el resultado.

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

Responda las siguientes preguntas:

1. ¿Qué es una *race condition* y por qué hay que evitarlas?

**Se llega a dar cuando varios threads empezando desde dos, pueden acceder a los datos compartidos e intentan cambiar la información de estos al mismo tiempo. Es necesario evitarlas porque puede causar mal funcionamiento de un programa. En ejemplo puede que el thread 1 realicé una acción y el resultado sea usado por otra acción, esa acción depende del resultado del thread 1. Pero viene el thread 2 y cambie el resultado entonces la acción consecuente será totalmente diferente a lo que resultaría del thread 1.**

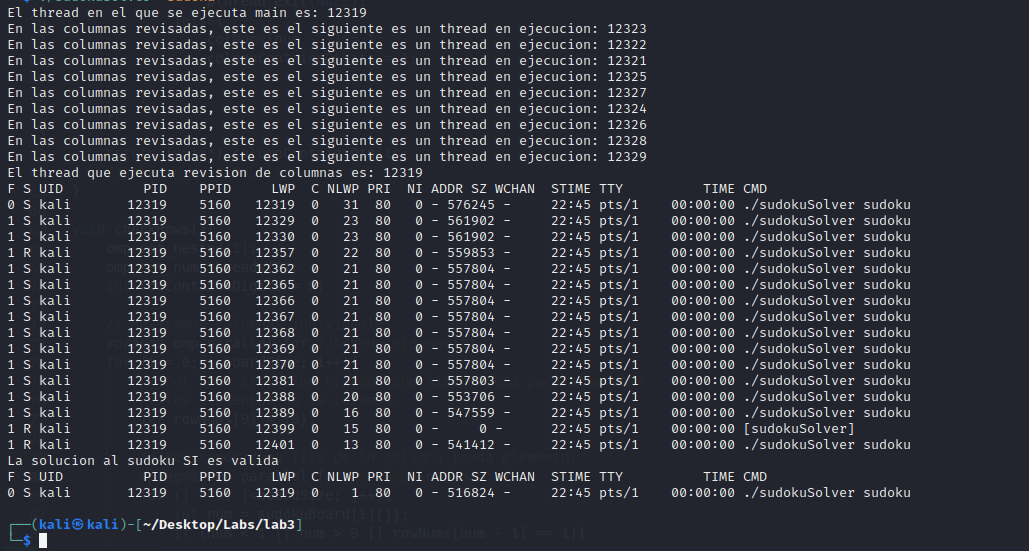
1. ¿Cuál es la relación, en Linux, entre pthreads y clone()? ¿Hay diferencia al crear *threads* con uno o con otro? ¿Qué es más recomendable?

**Esta relación consiste en que ambos crean nuevos subprocesos los cuales realizan lo que se les indique por lo tanto se relacionan en ese sentido. Su diferencia parte en que clone() crea un proceso basado en el proceso ya existente. Por otro lado, en la mayoría de casos es favorable utilizar Pthreads, debido a que al crear nuevos procesos utilizando clone()incluye mucho overhead agregado además de necesitar de un canal para poder comunicarse tales como son los sockets, archivos o memoria compartida.**

1. ¿Dónde, en su programa, hay paralelización de tareas, y dónde de datos?

**En el programa hay paralelización de tareas al momento de utilizar threads ya que se ejecutan varias tareas que se reflejan en el proceso principal. En la revisión de columnas y filas se puede notar la paralelización debido a es que este proceso se realiza en ocasiones repetidas durante toda la ejecución.**

1. Al agregar los #pragmas a los ciclos for, ¿cuántos LWP’s hay abiertos antes de terminar el main()y cuántos durante la revisión de columnas? ¿Cuántos *user threads* deben haber abiertos en cada caso, entonces? ***Hint*:** recuerde el modelo de *multithreading* que usan Linux y Windows.



Al agregar los pragmas for’s cuando se están validando las columnas hay 8 LWP’s, mientras que cuando está terminando hay 4. Deben haber 8 y 4 user threads respectivamente por el modelo 1 a 1 de multithreading de Linux

1. Al limitar el número de *threads* en main() a uno, ¿cuántos LWP’s hay abiertos durante la revisión de columnas? Compare esto con el número de LWP’s abiertos antes de limitar el número de *threads* en main(). ¿Cuántos *threads* (en general) crea OpenMP por defecto?

**En la ejecución y código ocurren 4 LWP que son los generados por OpenMP en el transcurso de la revisión de las columnas y los threads que se crean a partir del uso de OpenMP, son threads que se tienen a la disponibilidad dependiendo de la cantidad de procesadores. En este caso se le asignó 4 a la máquina virtual.**

1. Observe cuáles LWP’s están abiertos durante la revisión de columnas según ps. ¿Qué significa la primera columna de resultados de este comando? ¿Cuál es el LWP que está inactivo y por qué está inactivo? ***Hint***: consulte las páginas del manual sobre ps.

**Consultando la documentación, en la primera columna se indican los kernel threads y banderas del proceso. Revisando PS se nota que un thread se mantiene en un estado diferente a los demás durante el canal de wait(), esto indica que es un thread principal y se mantiene inactivo porque está bloqueado en estado fast space mutex.**

1. Compare los resultados de ps en la pregunta anterior con los que son desplegados por la función de revisión de columnas *per se*. ¿Qué es un *thread team* en OpenMP y cuál es el *master thread* en este caso? ¿Por qué parece haber un *thread* “corriendo”, pero que no está haciendo nada? ¿Qué significa el término *busy-wait*? ¿Cómo maneja OpenMP su *thread pool*?

**Es un grupo de threads donde todos son dirigidos por el master thread creado en líneas de pragma. Este mismo es el responsable de la revisión e imprime el texto indicado en el print dentro del programa, ya que los demás threads del grupo son copiados y reproducidos en un fork del master. En el tamaño del grupo se asigna una cantidad de threads ya configuradas, este ya indicado en el ejercicio del laboratorio. El término busy wait indica que el proceso chequea repetidamente la condición verificando si es True y también se utlizia para dejar agregado un tiempo de espera. OpenMP crea una cantidad indicada de threads y se crean en región paralélela para ser eliminados en un futuro en la última región.**

1. Luego de agregar por primera vez la cláusula schedule(dynamic) y ejecutar su programa repetidas veces, ¿cuál es el máximo número de *threads* trabajando según la función de revisión de columnas? Al comparar este número con la cantidad de LWP’s que se creaban antes de agregar schedule(), ¿qué deduce sobre la distribución de trabajo que OpenMP hace por defecto?

**Schedule(dynamic) de manera predeterminada funciona como FiFo y de esta manera distribuye el trabajo OpenMP.**

**Graphical user interface, text, application

Description automatically generated**

1. Luego de agregar las llamadas omp\_set\_num\_threads() a cada función donde se usa OpenMP y probar su programa, antes de agregar omp\_set\_nested(true), ¿hay más o menos concurrencia en su programa? ¿Es esto sinónimo de un mejor desempeño? Explique.

**Existe más ocurrencia ya que en la salida de PS despliega una mayor cantidad de LWP previstas en la revisión de las columas y al utilizar OpenMP se agrega Overhead ya que varios threads se comunican entre sí y es más lento (no en gran cantidad) porque el proceso es mutex y se realiza de esa manera para que no existan deadlocks incensarios, todo esto para seguridad del proceso.**

1. ¿Cuál es el efecto de agregar omp\_set\_nested(true)? Explique.

**Los números al momento de la revisión de columnas es diferente en los threads, comparado a cuando no se utilizó omp\_set\_nested. Anteriormente solo se podían observar set num threads la cual era la cantidad de threads desplegadas en PS que eran menores.**

**Deja el max-active-levels-var al número activo de niveles de paralelismo que soporta la implementación. Cabe mencionar que en la documentación de OpenMP esta rutina se considera obsolete.**